Development of a Real-Time Executive

Lab manual

Acknowledgement

Courses as complex as ECE350 and SE350 are not the work of a single person. This lab manual and the lab exercises contained within it are the products of many years of work by talented instructors, lab instructors, and teaching assistants. In particular, we would like to acknowledge the efforts of Irene Huang, Seyed Majid Zahedi, and Aravind Vellora Vayalapra, who contributed to the material on which this lab manual is based.

Chapter 1: Lab Administration

Groups

Group Size

The project is done in groups of four. Five is not allowed and three is not recommended. The workload is fixed regardless of the size of the group. All group members receive the same grade for each deliverable.

Group Sign Up

Learn is used for group sign up. The deadline for signing up will be communicated to you in the first week of classes. Note that grace days do not apply to group sign up. After the deadline, any student without a group will be randomly assigned to one.

Quitting Groups

Students can quit their group and join a new one only once, and only if the number of students in the new group does not rise above four. Students must notify the lab instructor in writing at least one week before the nearest lab deadline. The split-up will occur after that nearest deadline, so the group must continue to work together until the nearest deliverable is submitted. If a group member leaves the group, all members of that group lose the group sign up grade.

Source Code

Groups must maintain the source code and any other documents for their project in Gitlab. Gilab repositories will be created for each group, with group members added automatically. If a member leaves the group, they will be removed from your repository.

Collaboration

Explaining concepts to someone in another group, discussing algorithms/testing strategies with other groups, assisting other groups with debugging, and searching online for generic algorithms are allowed. Sharing test cases is also permitted, but groups are ultimately responsible for ensuring that the test cases they use are comprehensive and valid. One group cannot blame another if they rely entirely on other group's tests only to later find out that those tests are not sufficient to expose errors.

Sharing code with other groups, including by open-sourcing your code *even after this term*, is not allowed. Similarly, you may not use code from prior years if it has been made available contrary to this directive. Any suspected plagiarism or infractions of this honor code will be reported to the appropriate associate dean.

Lab Projects

Lab Equipment and IDE

For this lab project you will be using the STM32F401RE or STM32F411RE microprocessor packaged in an STM Nucleo-64 board. The boards are available at the UW bookstore and at online retailers and they are interchangeable. Students are expected to purchase their own board. Strictly speaking, each group

needs only a single board to complete the project. However, it is strongly recommended that all group members purchase their own board so that they can do development asynchronously.

The STM32Cube IDE will be used for automatic grading. It is essential that your Gitlab repository be organized in such a way that teaching staff can clone and run your code without reconfiguration in that IDE. You may use any IDE you wish for local development, but the STM32CubeIDE is fully featured, free, and designed specifically to work with the required board. If all group members use that IDE for development it is likely that your project will not encounter issues with autograding setup.

Deliverables

There are three major deliverables in the term. Groups must commit their code to Gitlab before the deadline for the code to be evaluated. During testing, teaching team members will clone the group's repository from the master branch and add additional testing files, run the group's code against those tests, and report the results. If the group continues to work on their code past the due date the latest commit before the due date will be used for testing.

Late Submissions

There are three grace days, including weekends, that can be used for late submissions without incurring any penalty. If your group intends to use them, you must contact the lab instructor with a commit number that you want graded. When all grace days are used, a 15% penalty per day or part day is applied. Submissions after three penalized days late are not accepted.

Code Testing Policy

The week after labs are due, teaching staff will test your code using private auto-grading files. Each test has a time limit of 30 minutes, although in the past most student code completed the test within 20 minutes. If your code crashes, hangs, or takes too long, marks will be deducted. Similarly, if your code is not set up to use the correct IDE, teaching team staff will not spend time converting it. Your project will receive a grade of 0.

Private test cases will not be released or discussed. Test cases will be re-used across multiple years, and it is expected that students thoroughly test their code before submission. If the teaching team determines that a very minor error is the cause of a large loss of grades (for instance, accidentally committing the wrong code when the correct code was written and unmodified after the due date), individual consideration will be given. Often this means that students may resubmit for a penalty to be determined by the lab instructor depending on the severity of the error.

Lab Repeating Policy

For students who are retaking the course, labs need to be re-done with new lab partners. Simply turning in the old lab code is not allowed. It is understood that the student may choose a similar route to the solution chosen last time the course was taken. However, it should not be identical.

Seeking Help

Discussion Forum

Piazza will be used for discussions. Students must ask any lab questions there, with the exception of personal questions. If you feel the need to post code or believe that your question may violate the sharing policies above, you may make a post to instructors only. Piazza will be set up so that posts may

be anonymous to the class but not to instructors. Teaching staff will ignore emails or tell you to post to Piazza if the email is not a private matter.

Office Hours

The lab instructor will hold office hours throughout the term at a time posted to Learn. TAs may also hold office hours at their discretion. Please note that teaching team staff are not expected to debug code.

Lab Facility

Although students may work locally with their own board, a lab room will be available for lab sessions and after-hours use. After-hours use will be granted by a door code that is unique to your class. The lab must be kept tidy, and no food or drink is allowed in the lab. If this is violated lab access will be revoked for the entire class. Note that you may be using the same lab as other classes, and it is possible that these other classes may reconfigure the computers while they are working on their own labs. It is expected that, if you sit down at a computer to work, you ensure that the computer is configured for your project. Groups cannot book a computer and keep it configured if they are not in the lab.

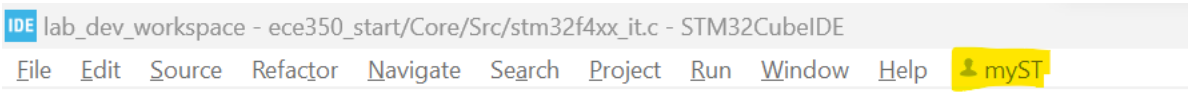
Chapter 2: Software Development Environment

The IDE

In this course we will be using the STM32CubeIDE for automated testing. We will attempt to use the most recent version *as of the start date of the course*, and will not change versions throughout the course except where crucial bug or security fixes are made by STM. The IDE may be installed locally, but it is available on lab computers in case your laptop does not communicate well with the board.

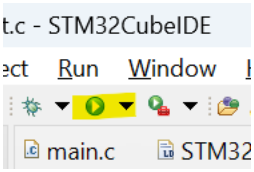
The IDE is available here: https://www.st.com/en/development-tools/stm32cubeide.html

The IDE is free, but you will need to sign up for a "myST" account. It is based on the Eclipse IDE so should be familiar to many of you. In order to ensure that all IDE functionality is available to you, you must log in to your myST account upon opening the IDE. The button to do so is at the top of the window:



Groups are free to use any other IDE they wish, but must be aware that automated testing will occur using STM32CubeIDE. Specifically, we will:

1. Open your STM project file (the *.cproj file)
2. Transfer the ae* files into your project
3. Click the "run" button:



Your project is then expected to compile and run, printing output to the serial port. If it does not do so but works on another IDE, it will still receive a grade of zero.

The IDE and Serial Communications

We need something on our computer that can read from a serial port. Luckily, STM32Cube has this built-in and it is remarkably smart.

1. If it isn't already open at the bottom of the screen, click Window->Show View->Console

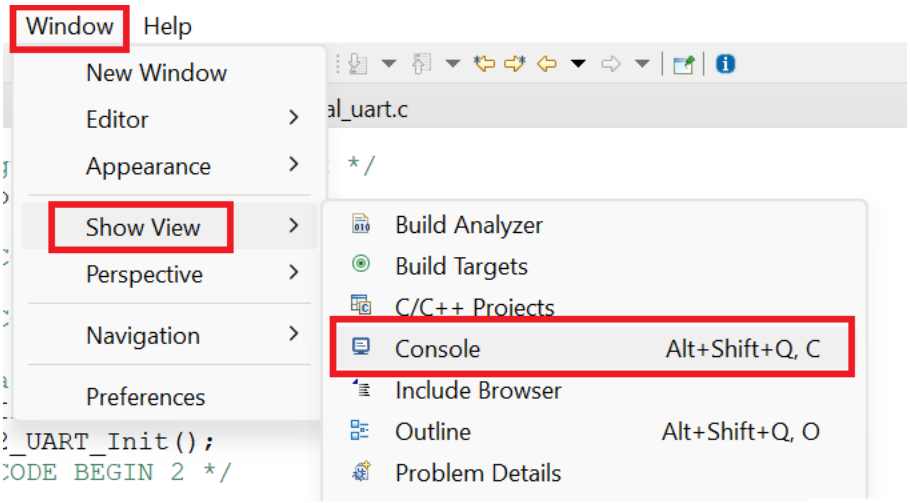


Figure 1: Opening the console

2. By default, this opens the Debug and Download console. We instead want the serial console. To do this we need to open a new console, so click the new console button, then select "Command Shell Console".

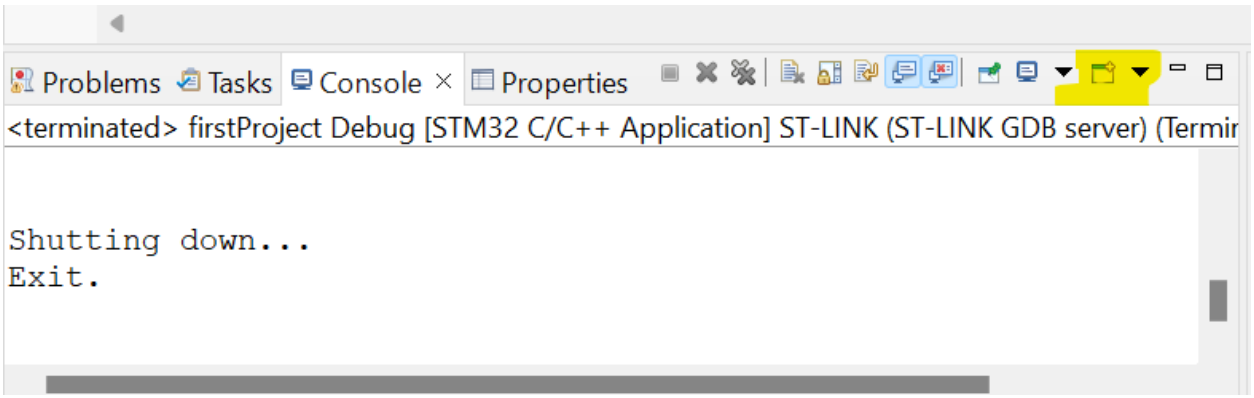


Figure 2: New console

3. Under Connection Type select Serial Port. If everything is connected properly, under Connection name select Cortex. Click "New…" to verify the settings. Your Serial Port number will likely be different from what is shown below, but otherwise the settings should be the same. If this is the case, name the connection something ("CortexM4" is fine) and click "finish", then click "OK" on the remaining window:

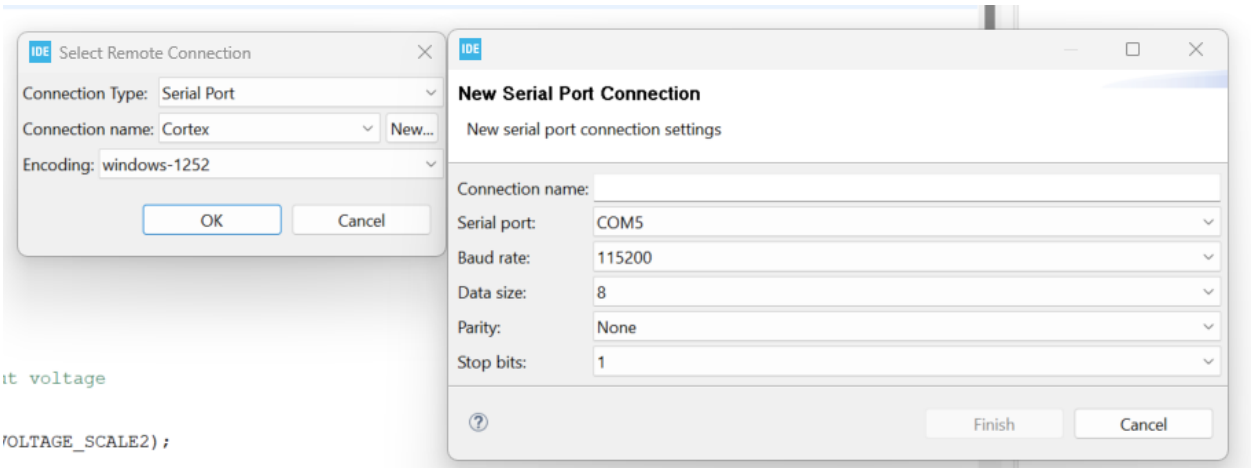


Figure 3: Choosing the serial port

4. This will open a new console that displays whatever the Cortex prints. Below is sample output from a simple program that simply prints the character “m” and then enters an endless loop:

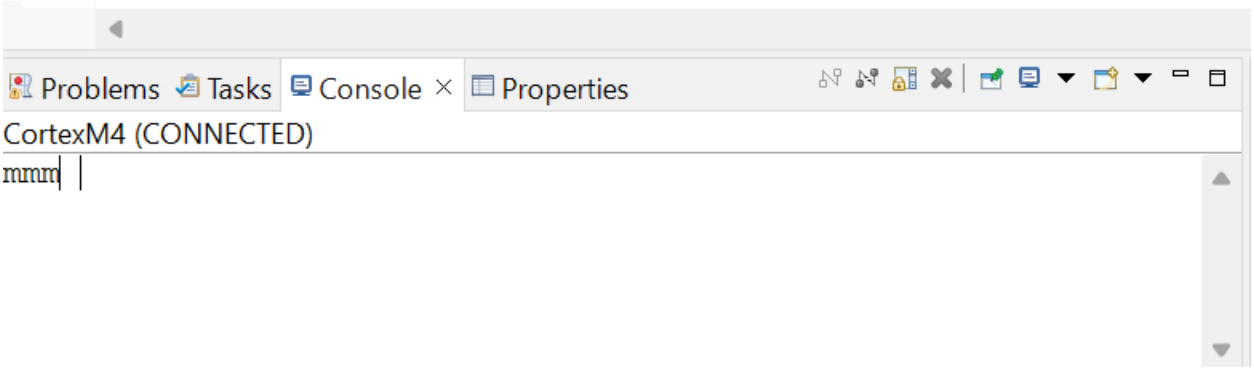


Figure:It's alive!

Setting up Gitlab

Each group is expected to maintain their source code using git. We will be using the University of Waterloo’s GitLab instance to manage git repositories, and we will use git to download your code during automatic grading. If you have not previously used GitLab, go to git.uwaterloo.ca and sign in with your UW credentials. This will create a git account for you.

You will automatically be added to a git repository once you have completed the group sign up task, before the first lab. This repository will contain all of the starter code. You should clone your group’s repository to get started.

Running the Starter Code

To run the starter code, open the folder that you have just cloned and double-click the `.cproj` file. This should open the starter code project in the STM32CubeIDE. If it does not, verify that you have correctly installed the IDE and cloned the repository. The, follow these steps:

1. Connect your STM32 Nucleo board to your computer and click the "run" button. You should see the build console at the bottom writing a number of messages to the screen. Crucially, somewhere in those messages should be the text `Download verified successfully`. This text means that the code compiled and is now downloaded onto the chip
2. Follow the steps above to set up your serial port. You should see the text "Hello, world!" printing repeatedly to the console

Understanding Automated Testing

Automated testing is performed via the files in the `ae*` set. For each deliverable, a set of public test cases will be released to you. These test cases will ensure that you have correctly written the API, but will not test edge cases. Note that these testing files are not included in the starter code itself because attempting to run them is expected to fail until the API is written. You should review the testing files as you work on your project, to ensure that you are not inadvertently misunderstanding crucial API functionality.

The starter code's `int main` function is not set up to run the automated tests. You must replace your development main.c file with the one provided with the automated tests before trying to get them to work. The reason we are doing it this way is so that you can have a clean main file to work on during your development. However, it means that no OS functionality can go into main.c, since it will be replaced during testing!

Chapter 3: RTX Overview

Introduction

Over the course of the term, you will implement a real-time executive (RTX) on the ARM Cortex M4 microprocessor. The specific chip we are using is the STM32F401RE (or STM32F411RE), which has 96KB of RAM and 512KB of on-chip flash memory. The chip also contains an on-chip UART, JTAG debug interface accessed via the STLink utility, and several on-chip timers.

The RTX will provide a basic multi-programming environment that supports priority (via deadlines), pre-emption, and dynamic memory management. The RTX is designed for co-operative, non-malicious software. That is to say, it will not be designed to consider security and it will be assumed that all test cases are written to conform to the API as it is written. The RTX will enforce a division between the kernel and user space.

Terminology

In lecture, you distinguish between programs, processes, threads, and tasks. In this lab manual we are going to use the terms “thread” and “task” only. A “thread” is the set of instructions that represents a single unit of concurrent programming. A “task” is a running instance of the thread, and includes all relevant in-memory context and variables, plus loaded, runnable instructions.

RTX Requirements

The RTX will be built in stages by your group. Each stage listed below has an associated deliverable. Each deliverable requires you to fully implement an API, specified in the chapters below.

- **Dynamic Task Management.** The RTX will support a fixed number of tasks, specified at compile-time. The RTX will support task creation and termination during runtime.
- **Dynamic Memory Management.** The RTX will support dynamic memory allocation using the first-fit memory allocation scheme.
- **Real-Time Scheduling.** The RTX will support pre-emptive scheduling using the Earliest Deadline First scheduler.

As you complete each deliverable, you will build on what you had before. That is to say, it will be necessary to have a working implementation of dynamic task management before you can begin working on dynamic memory management, because the memory management API will be written under the assumption that the task management API is working. Similarly, you must have a working implementation of both the dynamic task management and dynamic memory management APIs to get the real-time scheduling API working. A reference solution will not be provided, so it is up to the groups to ensure that the functionality is present, tested, and working for all lab projects.

RTX Testing and Coding

The automatic evaluation code is written under the assumption that your group has implemented the API exactly as shown. This means that you are free to add additional functions, data structures etc., but you are not free to change the API itself or to require automatic testing code to call these unspecified functions. As an example, it is expected that all automatic evaluation code initializes the RTX via a call to

the function `osKernelInit`, which is specified below. If your implementation instead requires the automatic evaluator to call some other function, it will not do so, and your tests are unlikely to succeed.

Global Restrictions

The RTX is expected to have a reasonably lean implementation. **No standard C library functions are allowed to be used in the kernel code**, with the exception of `printf`. Crucially, this means that you are not permitted to use `malloc` at all during this project (and in deliverable 2, you will be writing your own memory allocator anyway). If your group believes that a specific function from the C standard library is required, the group must write it themselves.

The API specifications below are not suggestions. Your group must ensure that the function prototypes and return values are exactly as specified.

The RTX will not support error recovery. It is assumed that application programmers deal with errors in their code.

Required Header Files

The automatic evaluation code assumes that the header files “common.h”, “k_task.h” and “k_mem.h” are written. Your group must decide what goes into each file. If the entire API is contained within only these three files, the automatic evaluation code should compile if there are no other errors in your code. For each project, you will receive a set of automatic evaluation files to ensure that your project will compile with the private test cases we will use. If you choose to write additional header files you must ensure that these files are included in one of the three required header files. The ae files will *not* be modified to accommodate your custom headers.

The expectation is that “common.h” is used to hold common definitions that multiple files may need. For instance, the maximum number of tasks your OS can support makes sense to be put into common.h. The functionality and definitions required for running threads should go into k_task, and the functionality and definitions for the memory allocator should go into k_mem. However, there are no specific marks allocated to whether you organize your files appropriately, so the decision is ultimately up to your group except where otherwise indicated in the lab manual.

Chapter 4: Multitasking Part 1 (Deliverable 1)

Objective

In these labs we work with an embedded development kit (STM Nucleo-64), that is equipped with a high performance and low power microcontroller (The STM32F401RE or STM32F411RE), which could be found in control applications ranging from coffee machines to robot arms.

Out of the box, the microcontroller can run application code that is essentially a single program, entering at the main() function of the application code. This is useful for simpler controllers (like in a coffee machine), where you could just use a big loop to cycle through tasks -- e.g. check for user button pushes, check the temperature & water level, and update the output power switches if any of these changed.

For more complex applications (like a robot arm), there could be multiple tasks with rather independent objectives, and the tasks could also vary in CPU time requirement depending on its current progress -- e.g. handle Bluetooth communications, run a gripping algorithm, check that the whole robot arm is error free. Doing these in one big loop would be extremely meticulous and limiting in flexibility. And having an RTX kernel on the microcontroller would greatly facilitate the application design process. The code for each task could then be implemented independently, as long as it is conscientious to yield when it's done. And the kernel would ensure the CPU is served to each of the tasks so that everyone gets a share.

In this project you will create a kernel and develop the multitasking features in your kernel using the scheme of co-operative multitasking and round-robin scheduling. Specifically, you will:

- Write the fundamental kernel API functions `osKernelInit`, `osCreateTask`, `osKernelStart`, `osYield`, `osTaskInfo`, `osTaskExit`
- Create the Task Control Block (TCB) data structure and set up relevant OS metadata as per your group's design
- Understand how the automatic evaluation code interfaces with your RTX, and run basic public tests to ensure that your implementation conforms to the API specification

Once this deliverable is complete, your RTX should be able to:

- Initialize relevant kernel data structures and other information
- Dynamically create and terminate tasks
- Run multiple concurrent tasks
- Obtain information about the tasks that are currently instantiated

Starter Code

In this project only we will provide you with starter code. This starter code is a bare-bones project that is organized in a way that is conducive to automatic testing and future expansion. It also provides an

implementation of printf. This code is provided to you in your group's Gitlab repository, and before moving on you must ensure you are able to compile, download, and run this code on your STM32 board.

Assignment

You will implement the following multitasking API as specified below.

Macros and Data Structures

Macros

For this lab, the relevant macros from the `common.h` file are as follows:

```
#define TID_NULL 0 //predefined Task ID for the NULL task
#define MAX_TASKS 16 //maximum number of tasks in the system
#define STACK_SIZE 0x200 //min. size of each task's stack
#define DORMANT 0 //state of terminated task
#define READY 1 //state of task that can be scheduled but is not running
#define RUNNING 2 //state of running task
```

Data Structures

An important data structure is the Task Control Block (TCB), which stores all relevant information required to run a particular task. Note that the definition of the TCB given below is considered a starting point. Your group may add to this definition as needed throughout the semester, but you may not remove anything from it.

```
typedef struct task_control_block{
      void (*ptask)(void* args); //entry address
      U32 stack_high; //start starting address (high address)
      task_t tid; //task ID
      u8 state; //task's state
      U16 stack_size; //stack size. Must be a multiple of 8
}TCB;
```

Design Considerations

Consider the following as a starting point to begin your design. Note that these suggestions are not comprehensive, but they should get you on the right track:

1. How are you planning on storing your TCBs?
2. How does the kernel know which task is currently executing?
3. How does the scheduler know which tasks are available for scheduling?
4. What data does the kernel need to store and access in order to load and unload tasks?

Function Specifications

Kernel Initialization Function

Prototype: `void osKernelInit(void)`

Description: this function initializes all global kernel-level data structures and other variables as required by your design. This function must be called before any other RTX functions will work. Since this function is used to set up your RTX, what it does and how it does it is largely up to your group. It is essential that everyone in the group understands and agrees to how this function works before attempting to write others.

Return value: none

Task Creation Function

Prototype: `int osCreateTask(TCB* task)`

Description: create a new task and register it with the RTX if possible. Once created, each task is given a unique task id (TID). A TID is an integer between 0 and N-1, where N is the maximum number of tasks that the kernel supports (including the null task), and is decided by the `MAX_TASKS` macro defined in the `common.h` file.

It is expected that the application code sets up the relevant TCB fields before calling this function.

Before returning, a successful call to `osCreateTask` updates the relevant kernel-level data structures with the information stored in the given TCB, and updates that TCB with the unique TID that has been allocated to that task.

Please note that the caller of this function never blocks, but it could be pre-empted.

Return value:

This function returns `RTX_OK` on success and `RTX_ERR` on failure. Failure happens when a new task cannot be created because of invalid input(s) or the state of the RTX. For example, the function returns `RTX_ERR` when the number of tasks has reached is maximum or the stack size in the TCB is too small (i.e., less than `STACK_SIZE`). There are other failure modes as well that will be addressed in future projects (for instance, if no memory exists to allocate a new stack).

Kernel Start Function

Prototype: `int osKernelStart(void)`

Description: This function is called by application code after the kernel has been initialized to run the first task. It is assumed that once the first task starts, subsequent tasks are run via various methods including yielding and pre-emption. Therefore, if this function is called more than once, or is called before the kernel is initialized, no action should be taken.

Groups will likely use this function to initialize the scheduling algorithm given how tasks are stored.

Return value:

This function returns `RTX_ERR` if the kernel is not initialized or if the kernel is already running. Otherwise, it will not return. The expected successful behaviour is for the kernel to start running tasks immediately.

Co-Operative Yielding Function

Prototype: `void osYield(void)`

Description: This function immediately halts the execution of one task, saves it contexts, runs the scheduler, and loads the context of the next task to run. The expected behaviour depends on whether the next task has already been running, or if it is its first time running after the kernel has been initialized.

If the next task has been running already, it should resume exactly where it left off. For example, if that task ran and then itself called `osYield`, the task should resume as though it had returned from this function. If the next task has not yet run, it should start from its entry point.

Groups implementing this function must ensure that, at least, the following steps occur:

1. The currently running task's context is saved onto that task's stack
2. The scheduler is run and the next task is selected
3. That thread's context is loaded from that task's stack
4. The CPU resumes execution of the next task

The context switch (saving and loading context) requires assembly code. Groups may elect to handle certain operations, such as running the scheduler, by calling C functions from within their assembly routines.

Return value:

None

Task Information Function

Prototype: `int osTaskInfo(task_t TID, TCB* task_copy)`

Description: this function receives two arguments: a TID and a pointer to a TCB. The function will retrieve the information from the TCB of the task with id TID, and fill the TCB pointed to by `task_copy` with all of its fields, if a task with the given TID exists. The automatic evaluation code will use this function to copy TCB data into a new TCB so that any further manipulations will not affect the existing TCBs in the OS. However, it should be noted that the application code is expected to ensure that the TCB pointed to by task exists and can be safely modified.

Return value:

This function returns `RTX_OK` if a task with the given TID exists, and `RTX_ERR` otherwise.

Function to Exit from Tasks

Prototype: `int osTaskExit(void)`

Description: this function, when called by a running task, immediately causes the task to exit and the scheduler to be called, much in the same way that `osYield` works. However, when this function is called the calling task is removed from the scheduler and any resources it was using are returned to the operating system as per your design for future use, including re-using the same TID and/or stack for subsequent tasks. For this deliverable this may be as simple as setting its status to `DORMANT`. However, for subsequent deliverables you will need to carefully consider things like memory allocation.

Return value:

This function returns a value of `RTX_OK` if it was called by a running task. Otherwise, it should return `RTX_ERR` and do nothing.

Chapter 5: Memory Management (Deliverable 2)

Overview

In this part of the project, you are going to write memory management functionality into your RTX. Tasks will have the ability to allocate and deallocate memory dynamically! This part builds upon your existing implementation, and you should start from your working Deliverable 1 code. Note that is acceptable, and even encouraged, to modify your Deliverable 1 code to accommodate the new functionality described in this part.

Objective

In this part of the project you will implement a First Fit memory allocation scheme and its associated API. Your API must be able to allocate memory if it is available, and deallocate memory as long as ownership rights are respected (that is, a task can free only its own memory, not that of any other task). Specifically, you will:

1. Implement a dynamic memory management system based on the First Fit allocation scheme using a freelist structure
2. Implement a memory-initialization function, which sets up your RTX and the microcontroller's memory
3. Implement allocation and deallocation functions, modifying your Deliverable 1 code if needed to implement the concept of memory ownership
4. Implement a utility function to analyze the efficiency of the allocation algorithm and its implementation

Background: Locating Free RAM

The idea behind creating our own memory allocator is that we identify an area of memory that is free – this will become our heap – and manage that memory by writing various important metadata to it, searching through that metadata to find free memory to allocate, and assigning memory to tasks by writing more metadata for each allocated block. The first challenge we have to overcome, however, is to locate that initial area of free memory. The problem is that the code we've written likely uses RAM to store certain pieces of information used by the image of our program, and we need to be careful not to overwrite our stack memory either.

The program whose job it is to determine where various pieces of information go in RAM is the linker, and it does so via *linker scripts*. The linker script for your project is in the project's main directory, and is called STM32F401RETX_FLASH.ld (or …F411… if you have that chip. The point is that it will have an "ld extension. If you see several such files, the one with "FLASH" at the end is what you are looking for). Open that file and you'll see that it's just a regular text file. You can explore it if you want, but there are only a few relevant lines for our discussion here today. Here, we will show you the lines as they would appear in the …F401… file. Yours may be slightly different.

1. **Line 39:** here, the symbol `_estack` is defined. This is the highest RAM address you can access.
2. **Line 42:** here, the symbol `_Min_Stack_Size` is defined. This is the size of the stack that the microprocessor starts out with. You will likely want to make sure it is at least `0x4000` (that is, add a zero)
3. **Line 47:** here, you see that the origin address of RAM is 0x2000000 (that is, a 2 followed by seven zeros, represented as a hexadecimal address). This is the lowest RAM address you can access. It's easiest to just use the numeric constant in your code, since this will never change and accessing this symbol in C code is a bit difficult
4. Following the definition of RAM, you will see a `SECTIONS` block. Go to line 174
5. **Line 174:** This is the end of your image in RAM. We want to define a symbol here. To do so, on the line immediately following this one, add the following code: `_img_end = .;` This instructs the linker to define a new symbol called "_img_end". This will resolve to an address in our C code that defines the start of free memory.

Accessing the Symbols in C

Note: the instructions in this section are just for learning purposes. You will need to remove all of this code from `main.c` and adapt it however you need for your specific RTX later.

The linker symbols defined in the ld script are accessible as external unsigned integers in C code. To view them, you may consider doing something like this in your main function:

```
...
extern uint32_t _img_end;
printf("End of Image: %x\r\n",&_img_end);
...
```

Notice that we had to access it via the ampersand (&) operator. This is because these symbols are defined by the linker and are strictly speaking not integers. The linker will resolve the symbols properly only if we access them in this way.

Determining your Heap Addresses

The start of your heap will be around `_img_end` (students often add a small offset of, say, 0x200 bytes, "just in case", which is fine though not excellent programming practice). The end of your heap will be at the location defined by the pseudocode: `_estack - _Min_Stack_Size`.

When allocating memory, never allocate to an address less than `_img_end` (or you will overwrite your other data, such as global variables) and never write to an address beyond `_estack - _Min_Stack_Size` (or you will start overwriting your stacks). Note that in the third deliverable we will relax this constraint, since our stacks will be dynamically allocated.

Will my Heap Ever be Overwritten?

These symbols tell you the size and location of various parts of your RTX's image in RAM as of compile-time, so it is natural to wonder whether the image can grow beyond these boundaries and corrupt the heap. The answer is complex.

RAM is organized in two distinct pieces. At the lower addresses (that is, closer to 0x2000000) you have the static portion of your image. This includes things like global variables, various startup data etc. This does not change as the program executes, so you can be confident that the top of your heap (that is, `_img_end`) remains constant. The bottom of your heap is a slightly different story. The bottom of your heap is also the last valid stack address. Assuming that everything is going well in your operating system and that tasks do not put more data on their stacks than they have stack space for, you may also assume that the bottom of your heap does not change.

In this course, we are assuming that all application code is non-malicious and bug free *during testing*, which means you do not need to consider the edge case of "what happens if a test case tries to overwrite my heap with a task's stack". However, during your own testing this may not be the case, so be attentive for this as you debug. Sometimes, your allocator may be working great but a task has a bug that causes it to overwrite the heap, making it look like the allocator is broken!

Background: Alignment and Metadata

Alignment

Whenever a task requests a block of memory, it must provide a size, in bytes, that it wishes to access. In this project, blocks can be any allocatable size on the Cortex M4. The M4 can access memory on four-byte boundaries only, so this means that the minimum amount of memory that can be allocated to a user is 4 bytes. However, this is an internal problem. From the application programmer's perspective, allocations of 1 byte should be allowed. Therefore, the memory allocator must *reserve* memory in four-byte aligned blocks, but allocations of smaller blocks should be allowed.

Practically what this means is that any time a task makes a request for, say, 1 byte of memory, at least 4 bytes are removed from the free memory pool. In our operating system, we will provide no additional protections against bad memory access, so it is assumed that the application programmer uses only the memory they have requested and assumes that any non-aligned memory is not accessible.

In addition, memory access of data types must be aligned, and this is again assumed to be the problem of the application programmer. What this means, for example, is that one cannot allocate 8 bytes of space but then try to store a 32-bit integer starting at the first byte. This would look like this:

```
uint8_t* eightBytes = k_mem_alloc(8);

uint32_t* badPointer = (uint32_t*)(eightBytes + 1);

*badPointer = 26;
```

The Cortex M simply cannot store a 32-bit integer on a non-four-byte aligned memory address, so this will cause a fault in the processor[1].

Metadata

Each block of memory, whether it is allocated or free, must have metadata associated with it. This metadata contains, for example, the size of the block, whether it is allocated or not, which task owns it

[1] We are telling you this now because *many* groups in the past did not respect four-byte alignment and encountered faults for *days* even though most of their memory allocator worked just fine!

etc. Exactly what goes into this metadata is up to you. The metadata is typically stored within the block itself, usually at the start (lowest address) of the block.

The metadata must be stored in memory, which means that the application program should not easily have access to it. For example, say that we are storing 8 bytes of metadata at the start of a block. The application program should receive a pointer to memory starting *after* the metadata (that is, for instance, if our metadata starts at address 0x100 and the metadata is 8 bytes long, the user application should receive the address 0x108 even though the metadata is allocated to that application as well).

The design of the metadata structure is an important one to figure out with your group before you start writing any code!

Assignment

Begin by reading and understanding everything in the "Background" sections above. In addition, it is strongly recommended that you make sure everyone in your group understands the concept of freelist memory and the First Fit algorithm.

Expectations of Efficiency

Your implementation is expected to be reasonably efficient. This means that groups must decide how they are going to handle coalescence during a free operation and how they are going to locate free memory reasonably quickly. Specifically, it is not acceptable to simply start from the beginning of memory and search the entirety of the heap every time memory is allocated or freed. Groups should consider design decisions, such as how to separate the free and allocated memory and how to traverse both sets of memory. It is understood that worst-case runtime for a freelist based memory allocator tends to O(N), where N is the number of blocks of memory. Algorithms with significantly worse runtime than this will not get full grades in the automatic grading.

Required Header File

You must write a header file with the exact name `k_mem.h` that contains all of the function prototypes and other definitions. This file will be included in the automated testing, so if your memory allocator works during your own testing but does not use this header to allow access to the memory API, it will receive a grade of zero.

Once you are certain you're ready, implement the following function specifications.

Function Specifications

Memory Initialization Function

Prototype: `int k_mem_init()`

Description: Initializes the RTX's memory manager. This should include locating the heap and writing the initial metadata required to start searching for free regions. As the manager allocates and deallocates memory (see `k_mem_alloc` and `k_mem_dealloc`), the memory will be partitioned into free and allocated regions. Your metadata data structures will be used to keep track of all of these regions.

Return value: Returns `RTX_OK` on success and `RTX_ERR` on failure, which happens if this function is called more than once or if it is called before the kernel itself is initialized.

Memory Allocation Function

Prototype: `void* k_mem_alloc(size_t size)`

Description: Allocates `size` bytes according to the First Fit algorithm, and returns a pointer to the start of the usable memory in the block (that is, after the metadata). The first-fit iteration should start from the beginning of the free memory region. The `k_mem_init` function must be called before this function is called, otherwise this function must return NULL. The `size` argument is the number of bytes requested. Any metadata must be allocated in addition to `size` bytes. If size is 0, this function returns NULL. The allocated memory is not initialized. Memory requests may be of any size.

By default this function assigns the memory region requested to the currently running task, or to the kernel if no tasks are running. This ownership information must be stored in the metadata, typically by storing the task ID of the task that allocated the memory, or some obvious value if the kernel allocated the memory.

Return value: Returns a pointer to allocated memory or NULL if the request fails. Failure happens if the RTX cannot allocate the requested memory, either for a reason stated above or because there are no blocks of suitable size available.

Memory Deallocation Function

Prototype: `int k_mem_dealloc(void* ptr)`

Description: this function frees the memory pointed to by `ptr` as long as the currently running task is the owner of that block and as long as the memory pointed to by `ptr` is in fact allocated (see below). Otherwise, or if this memory is already free, this function should return `RTX_ERR`. If `ptr` is NULL, no action is performed. If the newly freed memory region is adjacent to other free memory regions, they have to be merged immediately and the combined region is re-integrated into the free memory under management. This function does not clear the content of the newly freed region.

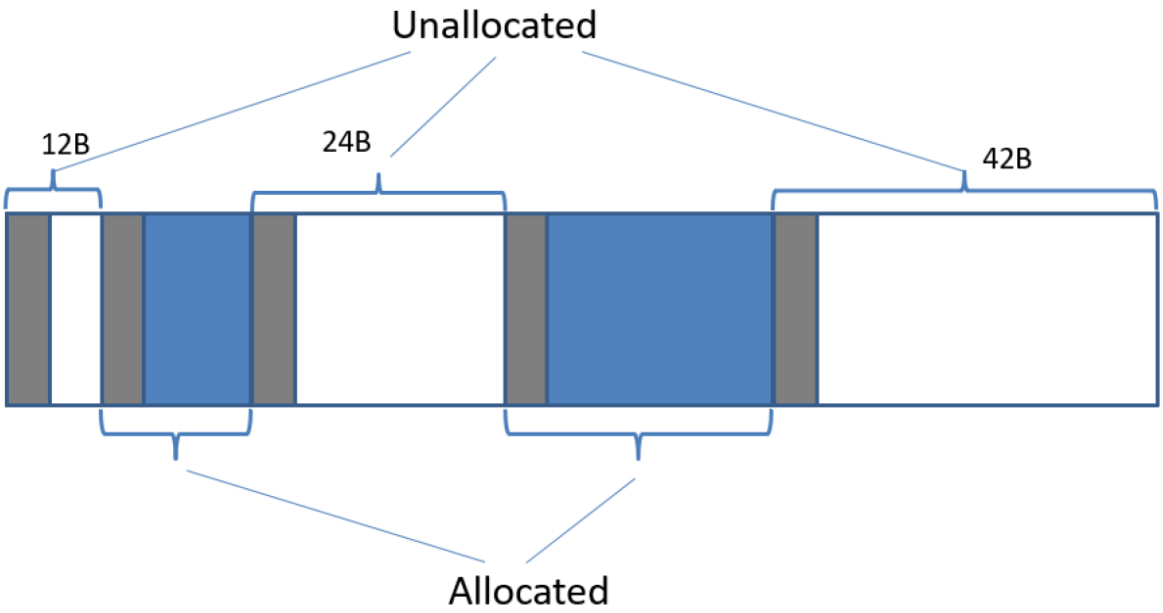
To determine if a memory region has been allocated, this function should examine the metadata immediately before the address `ptr`. You therefore must record in your metadata some way of determining this information. You may assume that, during testing, we will not do anything pathological. For example, we will not allocate 1000 bytes, then write valid metadata throughout that memory, and then attempt to free a random location. You may *not* assume that we will not attempt to free a random location in memory, though.

Return value: returns `RTX_OK` on success and `RTX_ERR` on failure. Failure happens when the RTX cannot successfully free the memory region for some reason (some of which are explained above).

Utility Function

Prototype: `int k_mem_count_extfrag(size_t size);`

Description: This function counts the number of free memory regions that are strictly less than `size` bytes. The space your memory management data structures occupy inside of each *free* region is considered to be free in this context, but not each *allocated* region. For example, assume that the memory status is as follows:



The grey regions are occupied by the metadata, the white indicates free space that can be allocated, and the blue regions indicate allocated memory regions. Calling `k_mem_count_extfrag` with 12, 42, and 43 as inputs should return 0, 2, and 3, respectively. If memory has not yet been initialized, return 0, as there are no free blocks of any size available.

Chapter 6: Pre-emptive Multitasking (Deliverable 3)

Overview

In this part of the project, you are going to expand your RTX so that it can perform pre-emptive multitasking. This means that we must introduce two important new pieces of functionality: timer-based interruption and priority. We will be using the SysTick timer, which is available on all ARM Cortex CPUs, to keep track of time. We will be implementing the pre-emptive Earliest Deadline First (EDF) scheduler, and assigning priority to tasks based on their deadlines.

Objective

Once this deliverable is complete, your RTX will be able to:

1. Dynamically allocate task stacks based on a desired size
2. Assign a priority to each task depending on its deadline
3. Pre-emptively switch a task if it has exhausted its time slice
4. Allow running tasks to change the priorities (deadlines) of others, potentially triggering a context switch
5. Allow tasks to sleep for a set period of time, during which they are not scheduled

Extremely important note: you should be starting this project from working Deliverable 2 code. All of the above points are *in addition* to what the OS can already do. This is a common point of misunderstanding – it is true that we are implementing pre-emption, but that does not mean that a task cannot also call the co-operative multitasking functions if it needs to. In addition, we are assuming that your memory allocator works!

Background: SysTick

All Cortex M chips have a timer on board called SysTick. SysTick is a timer that can be configured to trigger an interrupt, which is almost always called `SysTick_Hander`. SysTick works like any other timer you may have encountered: an integer gets loaded into a register, and every clock cycle that integer is decremented. Once the integer reaches zero it is reset and the associated interrupt is triggered. Theoretically one can change the period of the timer by modifying the integer, but the default value of 1ms set for us by STM32CubeIDE is sufficient for this project.

Locating SysTick_Handler

Setting up SysTick is quite challenging, but luckily in the starter code you received at the beginning of the term, it is set up for you. Presuming you have not modified the initial setup function calls then somewhere near the top of `int main`, you will see the following line:

```
SystemClock_Config();
```

This configures SysTick to trigger an interrupt once every millisecond. Specifically, the interrupt routine that is triggered can be found in the `Core->Src->stm32f4xx_it.c` file. Open that file and search for a function named `SysTick_Handler`. It should look like this:

```
void SysTick_Handler(void)
{
  /* USER CODE BEGIN SysTick_IRQn 0 */

  /* USER CODE END SysTick_IRQn 0 */
  HAL_IncTick();
  /* USER CODE BEGIN SysTick_IRQn 1 */

  /* USER CODE END SysTick_IRQn 1 */
}
```

Any modifications that you make to this function must be written *after* the line `HAL_IncTick()`. That function is used internally by various peripherals, for example UART, and if you remove or otherwise interfere with it many things will break!

You'll also notice that this function is a C function, rather than an assembly function. This is handy, since it is almost always easier to write things in C than assembly. Therefore, we will give you the following important advice:

When working with SysTick_Handler, assume that it is being called in a similar way to osYield – it is a C function that sets up and calls assembly code only when needed.

Suggested Exercise

In order to learn how SysTick works, it is recommended that you attempt a simple exercise. This is best done using a clean copy of the starter code, so that you can be sure that any errors are not due to your RTX interfering.

1. In `main.c`, declare a global integer
2. In `int main()`, initialize that integer to 1000
3. Declare that integer as `extern` in STM32F4xx_it.c
4. Every time `SysTick_Handler` triggers, decrement the integer
5. Whenever the integer is zero, print a short message using printf then reset the integer back to 1000

If everything was done correctly, you have just written a basic task that triggers once per second, rather than once per millisecond. The concept that you learned here is important – your tasks will have different deadlines. It is not possible to use one hardware timer per task, therefore you will need some other way of keeping track of those task deadlines and when they trigger.

Suggested Pre-lab: Pre-emptive Round-Robing Scheduling

Implementing the API for this lab requires you to modify your RTX so that it can handle pre-emptively switching tasks. To get you started, we recommend integrating SysTick into your existing co-operative scheduling framework first. This will require minimal modifications compared to the whole API, and it will give you a good idea of how to proceed with the more difficult parts. Attempt to implement the following functionality:

1. Every task should have a variable stored in its TCB. This variable should store the number of milliseconds remaining before this task must be switched out. We will refer to this as the task's "timeslice length"
2. Using SysTick, modify your RTX so that multiple tasks can loop forever without calling `osYield`. Whenever the timeslice of the currently running task expires, a context switch should be triggered and the round-robin scheduler should run
3. Verify that you didn't break anything by adding in another task that *does* call `osYield` as it runs, while the other tasks continue to run for their entire timeslice. It is crucial that this task also has a timeslice length value, and it is possible for it to take so long to run that it does not reach the call to yield during a single period

Point 3 above is crucial, and you need to be very careful. Consider the following scenario: Presume you have three tasks, A, B, and C, of which A and B are scheduled only by timeslice length and C is scheduled only by yielding. A race condition can occur where, in the middle of a context switch triggered by yielding, a second context switch is triggered because of timing out. Although this is a fairly rare occurrence, you must ensure that your OS does not fall victim to this race condition. If this happens, you will almost certainly hard fault, and it will appear randomly and be *very* hard to debug!

Modifications to Existing Codebase

This lab consists of two major parts – modifying the existing code so that it works with the new functionality, and adding a number of extra functions. In our experience, it makes more sense to modify the codebase first, then add the new functionality.

Modifying TCB (required)

- Modify your TCB so that all tasks have a deadline, in milliseconds
- You may consider modifying your TCB in other ways according to your own design

Additional task State (required)

- Add the SLEEPING task state with value of 3

Modifying Task Information (required)

- `osTaskInfo` should be modified so that anything you added to the TCB also needs to be copied in this function to the destination TCB

Modifying Task Creation: pre-emption and dynamic stack allocation

- `OsCreateTask` must be modified to allow for pre-emption and deadlines. This function now must create a task with a default deadline of 5ms. In the case where this function is called by a running task, if the newly created task has a sooner deadline than the currently-executing task, this function must trigger a task switch to the newly created task.
- This function should set the task's stack to the default value given by the macro `STACK_SIZE`, and dynamically allocate the stack using the memory management functions written in deliverable 2. The owner of the new stack should be the new task that is created.

Implementing EDF scheduling (required)

- You must replace your current scheduler with a pre-emptive EDF scheduler. When the EDF scheduler runs it must choose the task with the shortest deadline. Break ties by choosing the task with the lowest TID.

Modifying Kernel Initialization (depends on your RTX, but strongly recommended)

- `osKernelInit` should not take any input arguments. However, you should modify it so that any OS-level timer information is set up. Note that there is nothing specific here – that is, there is nothing we can test for. Rather, your RTX will be unlikely to work unless you set things like the round-robin timeslice or the TCB deadlines

Modifying Kernel Start (depends on your RTX, but strongly recommended)

- `osKernelStart` should not take any input arguments. However, you should modify it so that any relevant timers are reset. If you disabled interrupts at any time, you should enable them again etc.

Additions to Existing Codebase

Task Sleep Function

Prototype: `void osSleep(int timeInMs)`

Description: This function immediately halts the execution of one task, saves it context, runs the scheduler, and loads the context of the next task to run. In addition, this function removes this task from scheduling until the provided sleep time has expired. Upon awakening, a task's deadline is reset to its initial value. If no tasks are available to run (that is, all tasks are sleeping), the expected behaviour is for the NULL task to run until a task emerges from sleep.

Groups may notice that this function is very similar to `osYield`, and they would be correct. This function operates the same as `osYield` except the task cannot be scheduled until the sleep time is up.

Return value: None

Deadline Changing Function

Prototype: `int osSetDeadline(int deadline, task_t TID)`

Description: This function sets the deadline of the task with Id `TID` to the deadline stored in the `deadline` variable. Any task can set the deadline of any other task. This function must block – that is, when it is called, the OS should not respond to any timer interrupts until it is completed. It is possible for this function to cause a pre-emptive context switch in the following scenario: presume task A is currently running and has a deadline of 5ms. task B is not running and has a deadline of 10ms. If task A calls this function to set task B's deadline to 3ms, task B now has an earlier deadline and task A should be pre-empted.

Return value: this function should return RTX_OK if the deadline provided is strictly positive and a task with the given TID exists and is ready to run. Otherwise it must return RTX_ERR.

Task Creation with Deadline Function

Prototype: `int osCreateDeadlineTask(int deadline, int s_size, TCB* task)`

Description: Create a new task and register it with the RTX if possible. This task has the deadline given in the `deadline` variable and stack size given by the `s_size` variable, which is assumed to be in bytes. Once created, the new task is given a unique id (TID). A TID is an integer between 0 and N-1, where N is the maximum number of tasks that the kernel supports (including the null task), and is decided by the `MAX_TASKS` macro defined in the `common.h` file.

It is expected that the application code sets up the relevant TCB fields before calling this function.

Before returning, a successful call to `osCreateDeadlineTask` updates the relevant kernel-level data structures with the information stored in the given TCB, and updates that TCB with the unique TID that has been allocated to that task.

The caller of this function never blocks, but it could be pre-empted. Specifically, in the case where the calling task has a longer deadline than the newly created task, a pre-emptive context switch must occur.

This function must dynamically allocate a stack of the given size using the memory management functions from deliverable 2. The owner of the new stack should be the newly created task.

Return value: this function returns `RTX_OK` on success and `RTX_ERR` on failure. Failure happens when a new task cannot be created because of invalid inputs or if the RTX cannot allocate a new task. Invalid inputs include the same cases as discussed in `osCreateTask` for deliverable 1, as well as the deadline being zero or strictly negative or the stack size being less than `STACK_SIZE` or too large given the amount of free memory available.